MOFScript User Guide

Version 0.9 (MOFScript v 1.4.0)
February 2011
Author: Jon Oldevik

Table of contents

1 Introduction 4
2 Feature Overview 4
3 MOFScript within Eclipse 4
3.1 Code completion 5
3.2 Compiling MOFScript files 6
3.3 Executing MOFScript files 6
3.3.1 Executing with multiple input models 7
3.3.2 The generated files 8
3.4 Default repository for MOFScript 9
3.5 Preference page 10
3.6 File properties 11
3.7 File popup menu 11
3.8 New file wizard 13
4 The MOFScript Language 13
4.1 Texttransformation 13
4.2 Imports 14
4.3 Entry point rules 14
4.4 Rules 15
4.4.1 Return values 15
4.4.2 Parameters 16
4.5 Properties and variables 16
4.6 Built-in types 16
4.7 Files 17
4.8 Print statements 17
4.9 Escaped output 17
4.10 Iterators 18
4.11 Conditional statements 19
4.12 While statements 19
4.13 Select expressions 20
4.14 Logical Expressions 20
4.14.1 Expression grammar 20
4.15 Transformation inheritance 20
4.16 Abstract rules 21
4.17 Rule overriding 21
4.18 Invoking External Java Methods 22
4.19 Reflection operations on model elements 22
4.19.1 Reflection ops for EObjects 22
4.19.2 Reflection ops for EClasses 23
4.19.3 Reflection ops for EClassifiers 23
4.20 Built-in operations 23
4.20.1 String operations 23
4.20.2 Integer operations 25
4.20.3 List Operations 25

4.20.4 Hashtable Operations 26
4.20.5 PropertyMap Operations 26
4.20.6 Model Collection Operations 26
4.20.7 Model Operations 27
4.20.8 OCL Operations 27
4.20.9 System / utility operations 27
4.20.10 UML2 Operations 28
4.21 Unprotected Blocks 28
4.22 MOFScript Aspect extension 29
4.22.1 The aspect 29
4.22.2 Pointcuts 29
4.22.3 Advice 30
4.22.4 Executing the aspect 30
4.23 Model to Model Transformation 30
4.23.1 Declaring output parameters 30
4.23.2 Using output models from transformations (in Java) 31
4.23.3 Creating instances – the *new* operator 31
4.23.4 Setting property values 31
4.23.5 Adding values to collection properties 31
4.23.6 Cloning an object 32
4.23.7 Storing an object 32
4.23.8 Obtaining and storing an EResource 32
5 The MOFScript debugger 32
6 Integrating with MOFScript Java API 33

1 Introduction

The MOFScript tool is an implementation of the MOFScript model to text transformation language. MOFScript was submitted to the OMG process for MOF Model to Text Transformation (http://www.omg.org/cgi-bin/apps/doc?ad/05-11-03.pdf).

MOFScript is as an Eclipse plug-in, but can also be run as a standalone Java application. It supports parsing, checking, and execution of MOFScript transformations.

This document is intended as a user guide for MOFScript and covers the basics of the tool and the language facilities.

2 Feature Overview

MOFScript supports the following features:

Model to text transformations

Model to model transformation

Traceability

Multiple file output

Multiple model output

Light weight debugging

Configurable output encoding

Support for reading and writing property files

(Transformational Aspects)

3 MOFScript within Eclipse

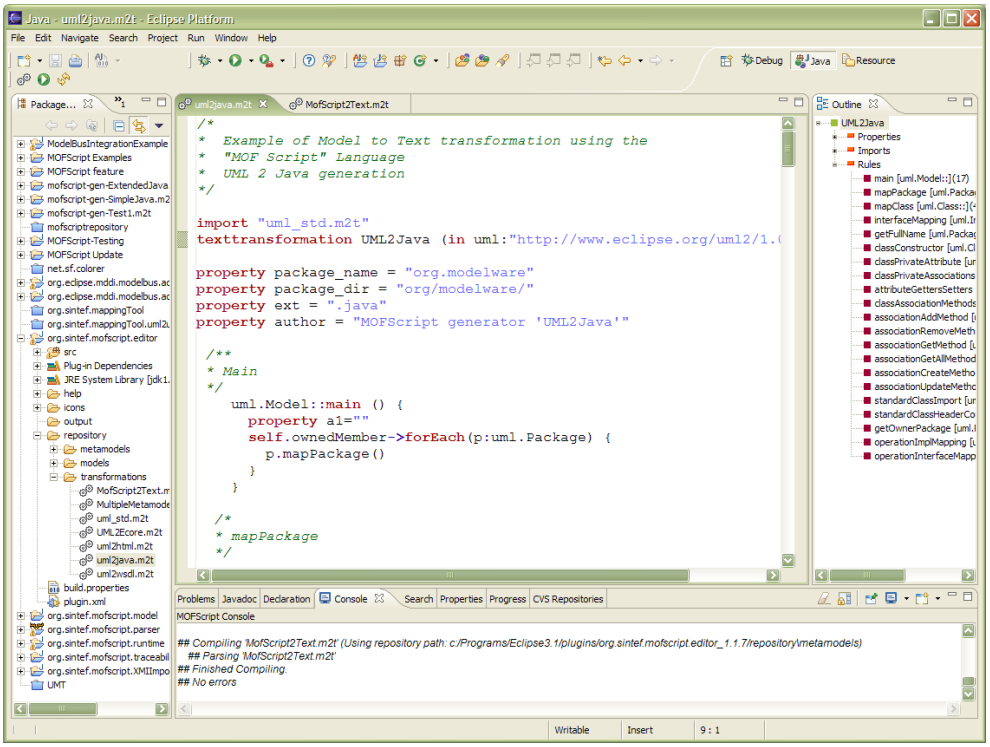
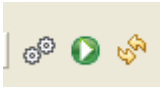


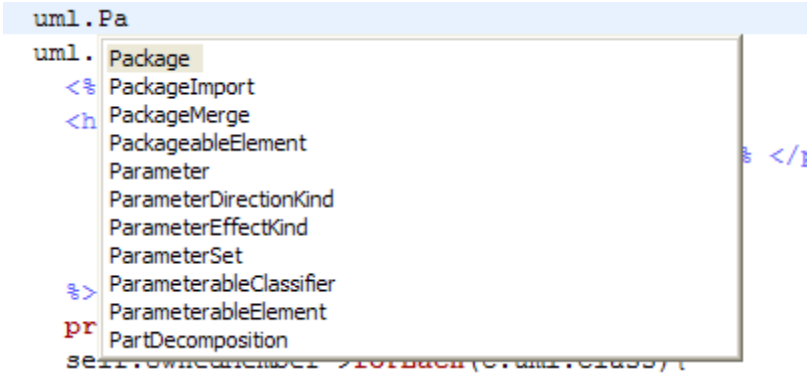
Figure 1 Eclipse after opening a MOFScript file

In the Eclipse UI, three action buttons will appear that provides the basic functionality of the MOFScript tool: *Compiling and Executing.*

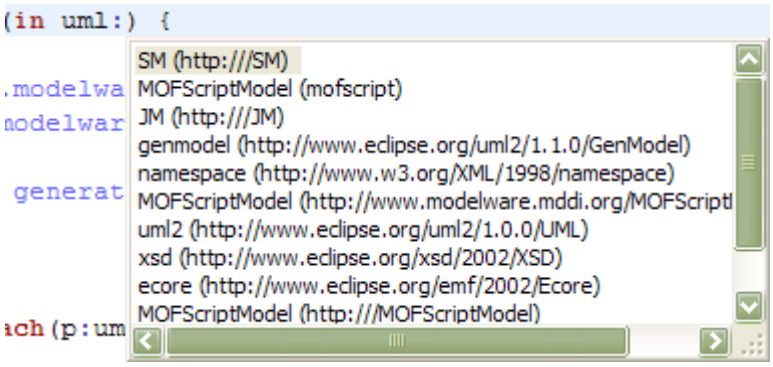


3.1 Code completion

The MOFScript editor supports code completion. Code completion is activated for metamodel or variable references.



The completion selector for selecting input metamodels for a transformation is shown below. It shows all packages registered in Eclipse EMF as well as those located (as files) in the metamodel file directory.



3.2 Compiling MOFScript files

The MOFScript code is compiled (parsed and checked) either by

(a) Changing content and saving it or

(b) By pressing the 'Compile' action button .

Compilation of the MOFScript code will initiate a compile process which parses the text, creates a MOFScript model, and checks it for errors (syntactical and semantic errors)

Errors are presented in the 'Problems' pane of Eclipse. They can also be seen in the 'Console' pane of Eclipse.

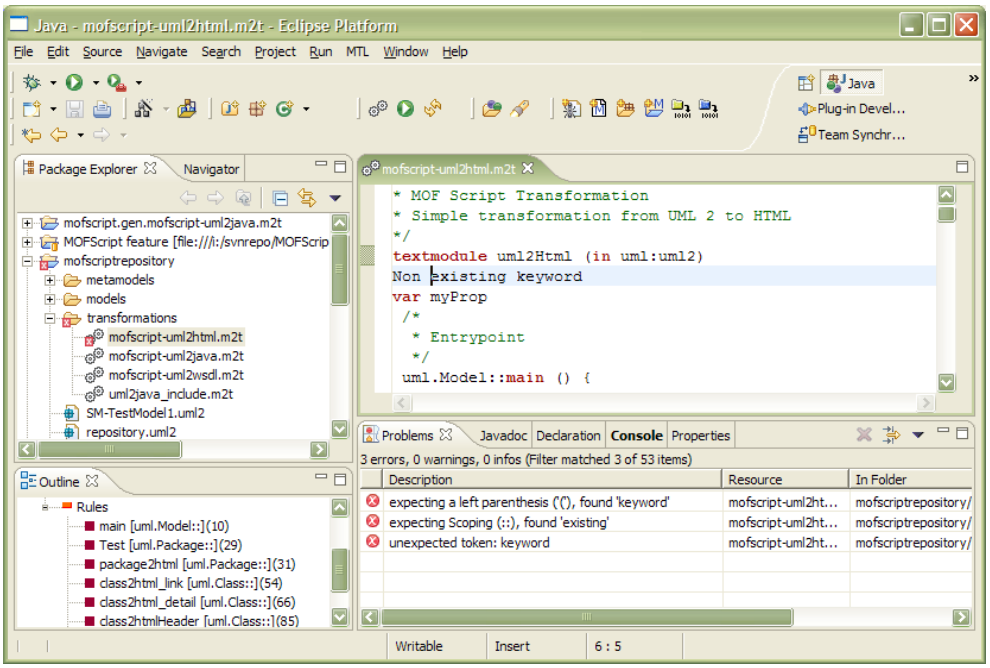


Figure 2 Error presentation

3.3 Executing MOFScript files

Execution of scripts can only be done if the scripts are free of errors. A compilation is always done (by the tool) prior to execution.

The result of a transformation is normally a set of files, generated to some location on the file system.

(The Eclipse 'Console' pane prints what is generated in terms of output files).

Execution is started with the execution action button or the 'execute previous transformation button' ( ).

The first time a transformation is executed, the user must select an input file. A file selection dialog is launched, where the user selects the source file for the transformation.

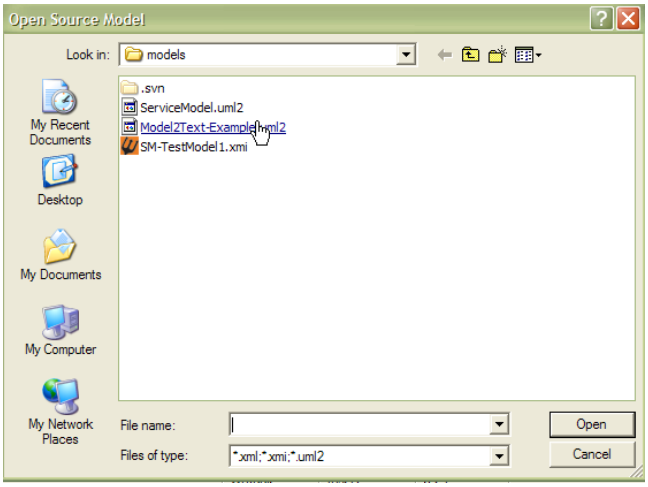


Figure 3 File open dialog for selecting input model

Input model selection can be chosen from Eclipse workspace – or from the file system. A preference in the preference manager 'Load files from workspace' defines the preferred way of opening files. The default is *load from workspace*.

3.3.1 Executing with multiple input models

If a transformation has several input model declared, the user must provide several input models when executing the transformation. For example, in the transformation example below, two input model parameters are given.

```
texttransformation MultipleMetaModels (in
uml:"http://www.eclipse.org/uml2/1.0.0/UML", in
ec:"http://www.eclipse.org/emf/2002/Ecore") {

  main () { // can also use module::main, which is the same
    uml.objectsOfType (uml.Class)->forEach (cl) {
      cl.ecoreModelTest()
    }
   // '\n Looking for ecore objects'
    ec.objectsOfType (ec.EClass)->forEach (eccl) {
      eccl.umlModelTest()
    }
  }

  ec.EClass::ecoreModelTest () {
```

```
    stdout.println ("Ecore class: " + self.name)
  }

  uml.Class::umlModelTest(){
    stdout.println ("Class: " + self.name)
  }

}
```

The user will be prompted two times to open model files.

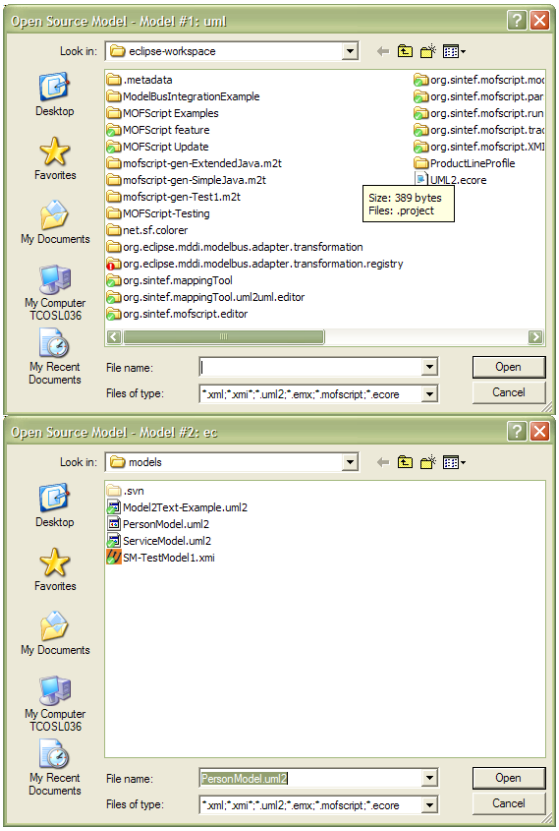


Figure 4 Opening multiple model files.

3.3.2 The generated files

The files generated are places on the file system where at a user specified location (specified by the *file* statements).

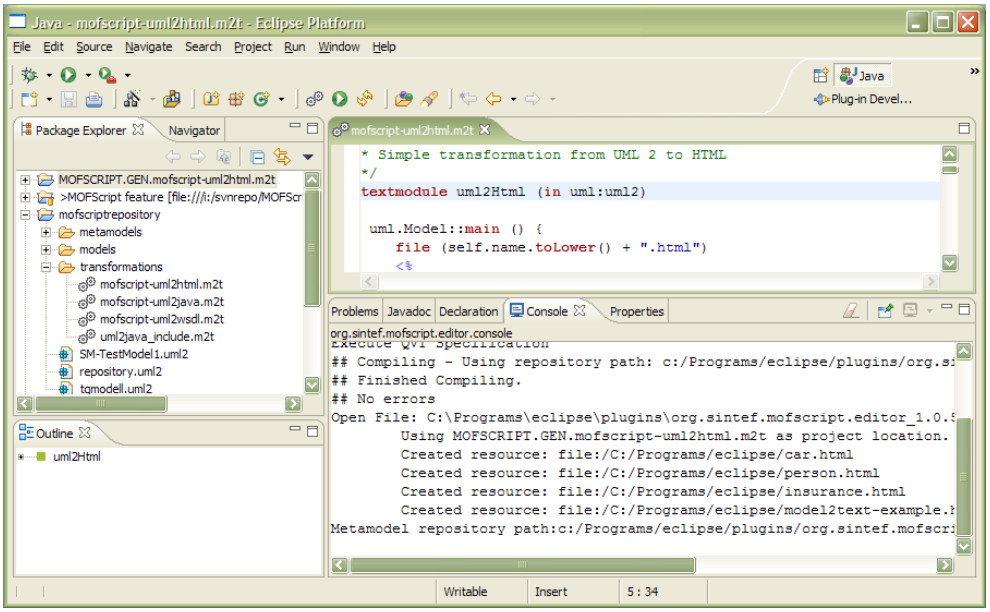


Figure 5 Executing transformation – the output

In the example in Figure 5, the transformation specifies file output to be model element name + "html", which is shown in the 'Console' during execution. In addition, a project is generated (if it does not exist), where links to the generated files are created (Figure 6)

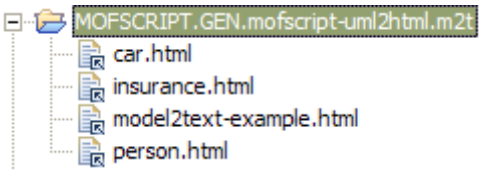


Figure 6 Generated project and file links

3.4 Default repository for MOFScript

The MOFScript tool uses two logical repositories for locating metamodels. It uses the built-in global repository in Eclipse/EMF and it uses a file-based location which can be configured by the user.

The file-based metamodel repository path is (by default) determined by the installation path of the MOFScript plugin (the editor plugin) + a named repository path ("*repository*").

For end users, this directory is visualized as an Eclipse project called '*mofscriptrepository*'. If you check under 'mofscriptrepository/metamodels', you will find the metamodels visible for the tool. This repository also contain some example models and transformation. This will be apparent for you as a user when you write a new MOFScript transformation, or change the input metamodel.

As seen in Figure 7, the set of available metamodels pops up when the user is defining the *texttransformation* / *textmodule* with parameters.

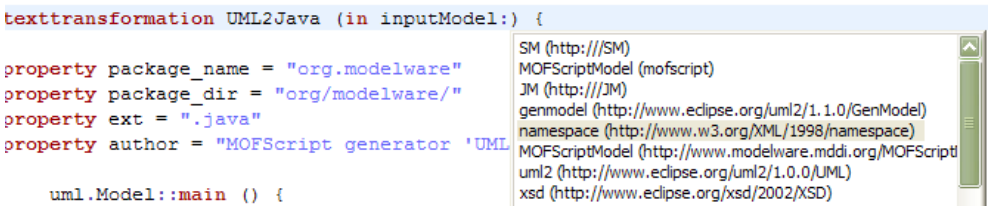


Figure 7 Selecting the metamodel for the transformation

To identify the metamodel is default used the URI of the registered metamodel. The name may also be used.

Metamodels that are registered in the global Eclipse/EMF package registry is preferred over file-based metamodels, in case a URI/name exists both places.

3.5 Preference page

The preference page for MOFScript is available under Eclipse preference pages

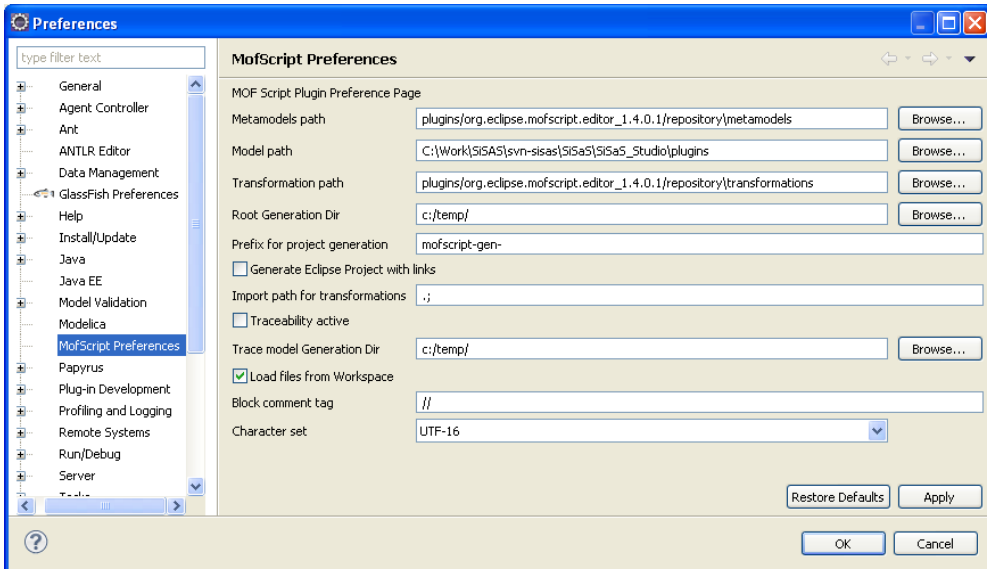


Figure 8 Preferences for MOFScript

(*Windows/Preferences menu) (*Figure 8*).*

The following preferences can be controlled by using the preference settings:

- **Metamodel path**: Defines the path from which the MOFScript tool fetches the metamodels for transformations.
- **Model path**: This path is used by the editor as the default location to look for input models.
- **Transformation path**: This path is currently not used.
- **Root generation directory**: This property is used to determine the location of output files. It is only used for output that is allocated with relative path.
- **Prefix for project generation**: Controls the prefix given to generated Eclipse project (the default value is *mofscript-gen).*

- **Generate project**: A boolean value which determines if an Eclipse project should be generated or not.
- **Import path**: Import path used for transformation parsing. This should be a list of directories separated by semi colon.
- **Traceability active**: Turns on / off traceability generation. Currently, the implementation of traceability is not finalized. Turning this on will now only result in some print messages.
- **Trace model generation dir**: Where to store the tracemodel.
- **Block comment tag:**
- **File loading preference:**
- **Encoding:**

3.6 File properties

Each MOFScript file has a set of MOFScript properties specific to that transformation file, which can be changed by the user (Figure 9)

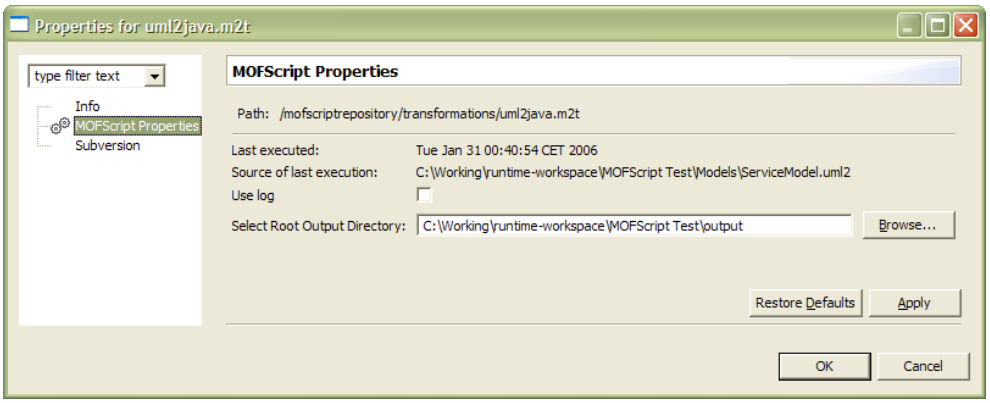


Figure 9 MOFScript properties

The properties currently available are:

- **Use log**: Turns on/off the logger for the transformation. If **on**, the log statements in MOFScript (*log ("this is a log message"))* will be printed. If **off**, the log statements are ignored.
- **Select Root Output Directory**: Sets the root directory property which will be used for this particular transformation. If not set, the global property will be used.

3.7 File popup menu

Some MOFScript actions are available on the popup menu on MOFScript files (Figure 10).

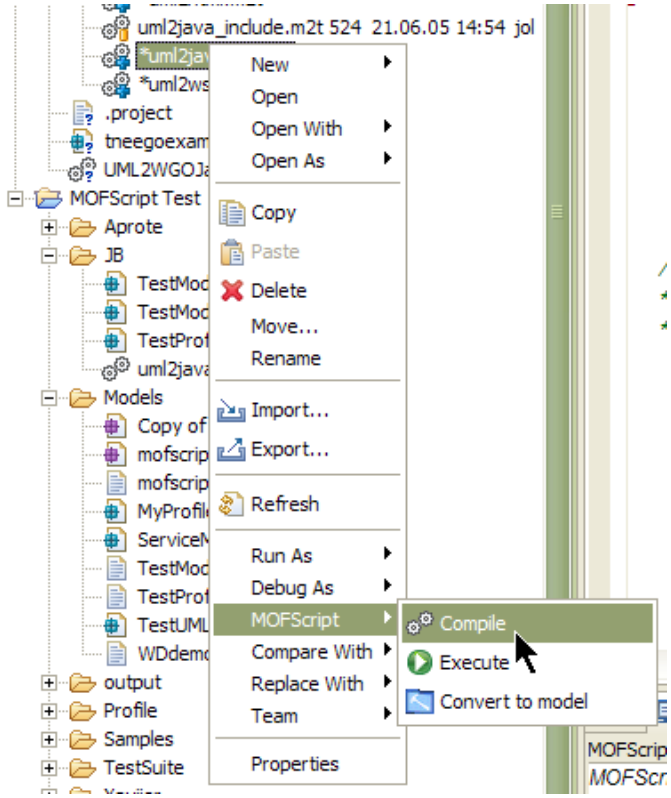
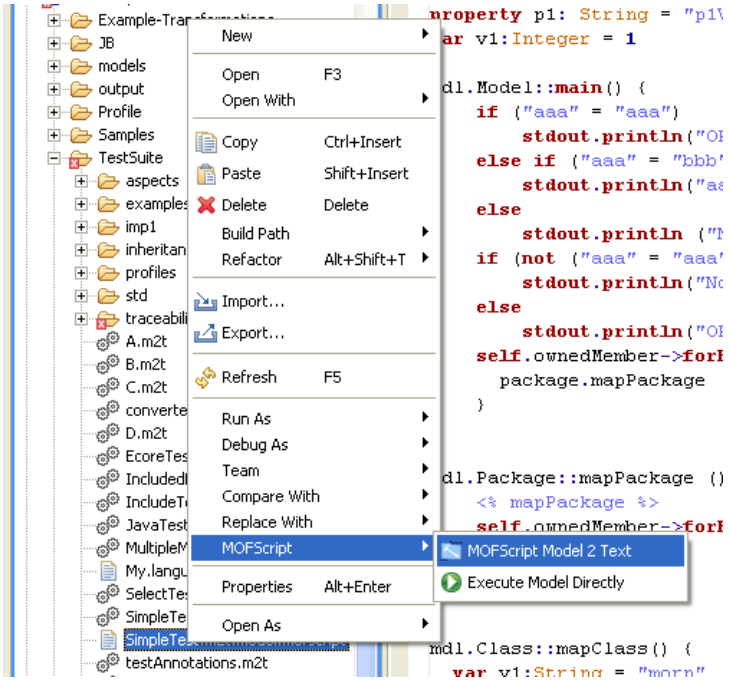


Figure 10 MOFScript file popup

Some other actions are available for mofscript model files (with .mofscript extensions), namely converting the model to mofscript text or executing the model directly.



Figur 1 MOFScript model popup actions.

3.8 New file wizard

In order to create a new MOFScript transformation from scratch, the MOFScript file wizard can be used (Figure 11).

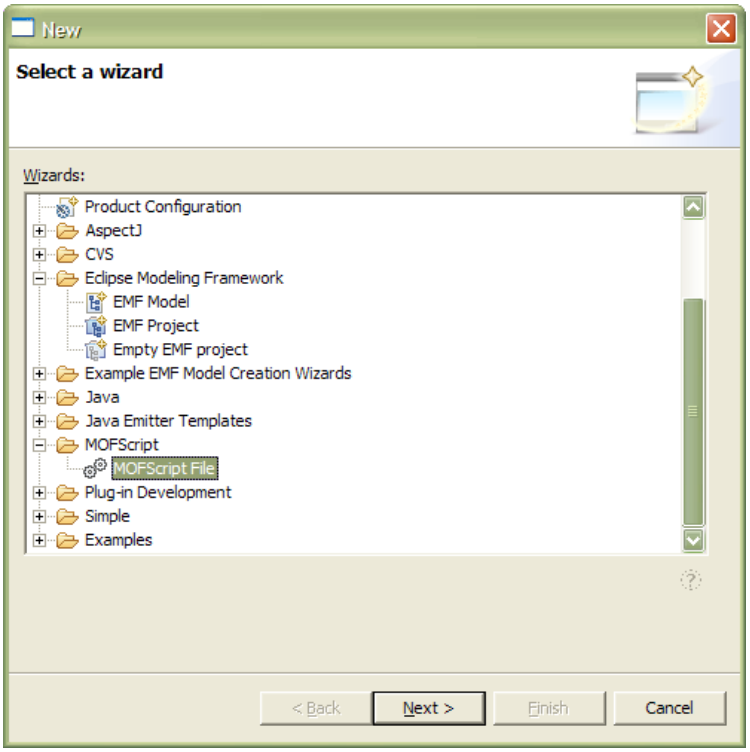


Figure 11 MOFScript file wizard

Using it will create a new skeleton transformation as a starting point.

4 The MOFScript Language

This section describes the various MOFScript language constructs.

4.1 Texttransformation

A Texttransformation defines the name of the module, which can be any name chosen, independent of file name. (Either of the keywords *texttransformation* or *textmodule* can be used.)

It defines the input metamodel in terms of a parameter.

```
texttransformation testAnnotations (in
                        uml:"http://www.eclipse.org/uml2/1.0.0/UML"))
```

A texttransformation may have several input model parameters. These should be separated by comma.

```
texttransformation TransformationWithSeveralMetaModels (in
uml:"http://www.eclipse.org/uml2/1.0.0/UML", in
ec:"http://www.eclipse.org/emf/2002/Ecore") {
```

```
}
```

4.2 Imports

A transformation module may import other transformations as libraries or for extension.

This is done using the *access library, access transformation,* or *import* keywords. (Currently, the semantics of these are the same. There are different syntax flavors to select from, as illustrated below. The simples one if just *'import "a-file.m2t"'* . The names given to an import is currently not used.

```
import aSimpleName ("std/stdLib2.m2t")
import "std/stdLib2.m2t"
```

Imported transformations are fetched by the parser, looking first in the *current directory,* then in any directory specified by the *import path*. In the preference manager, the import path property can be defined. It should be on the form *path1;path2;path3*. The parser will look for imported transformations in each path given if not found.

4.3 Entry point rules

Entry point rules defines where the transformation starts execution. It is similar to a Java main.

It may have a context (in the example uml.Model), which defines what metamodel element type that will be the starting point for the execution. Its body contains statements.

```
uml.Model::main () {
  self.ownedMember->forEach(p:uml.Package)
  {
    p.mapPackage()
  }
}
```

An entry point may have a context type with several instances. In that case, the entry point will be executed for each instance of the type. An example is shown below, where the entry point context type is the *uml.Class*.

```
uml.Class::main () {
  'class: ' self.name
}
```

Rules can be specified without context type, so also entry point rules. An entry point with no context type will be executed once. It is specified using the *module* keyword or without any keyword. To retrieve model input using this approach, the model parameter to the transformation must be used, combined with the operation *'objectsOfType'* to retrieve contained model objects.

```
module::main () {
    uml.objectsOfType (uml.Package)
}
```

or

```
main () {
    uml.objectsOfType (uml.Package)
}
```

The *objectsOfType* operation may be applied to any model element to retrieve a set of contained objects of a certain type.

4.4 Rules

Rules are basically the same as functions. They can have a context type, which is a metamodel type. They may also have a return type, which may be a built-in type (4.6) or a model type. The body of a rule contains a set of statements.

```
uml.Package::mapPackage () {
  self.ownedMember->forEach(c:uml.Class)
     c.mapClass()
}


uml.Class::mapClass(){
    file (package_dir + self.name + ext)
    self.classPackage()
    self.standardClassImport ()
    self.standardClassHeaderComment ()

    '
    public class ' self.name ' extends Serializable { '
       self.classConstructor()
       '
       /*
        * Attributes
        */
    '
       self.ownedAttribute->forEach(p : uml.Property) {
             p.classPrivateAttribute()
       }
       newline(2)
   '}'
}
```

4.4.1 Return values

A rule may also return a value, which can be reused in expressions in other rules. To return a value, the *result statement* is used.

```
uml.Package::getFullName (): String {
  if (self.owner != null)
    result = self.owner.getFullName() + "."
  else if (self.ownerPackage != null)
    result = self.ownerPackage.getFullName() + "."

  result += self.name.toLower().replace(" ", "_");
}
```

A return value may also be given by a 'return' statement, which immediately terminate rule execution and returns with the given value (if any)

```
uml.Package::getFullName (): String {
     result self.owner.getFullName();
}
```

4.4.2 Parameters

A rule may have any number of parameters. A parameter can be of a built-in type (4.6) or a metamodel type.

```
uml.Model::testParameters2 (s1:String, i1:Integer) {
  stdout.println("testParameters2: " + s1 + ", " + i1)
}

uml.Model::testParameters3 (s1:String, r2:Real, b1:Boolean,
package:uml.Package) {
  stdout.println("Package:" + package.name)
  stdout.println ("testParameters3: " + s1 + ", " + r2 + ", " + b1 +
" " + package.name)
}
```

A rule may be without context type. This is declared by using the keyword *module* or omitting a context type altogether (see section 4.3).

4.5 Properties and variables

Properties and variables can be defined either globally or locally within a rule or a block (e.g. iterator block).

A property is a constant, which is assigned to a value on declaration. The type of a property can be any of the types in 4.6, a model type, or it can be untyped in the declaration. Its type will then be determined by the value assigned.

A variable can change its value during run time in assignments. A variable can be type by any of the types in 4.6. It may also be defined untyped in the declaration. Its type will then be determined by the value assigned. If no type is assigned, its type will become a 'String'.

```
property packageName:String = "org.mypackage"
var myInteger = 7
```

4.6 Built-in types

The built-in types in MOFScript are summarized below

- **String**: The string type, which represents text values.
- **Integer**: The integer type;
- **Real**: The real type;
- **Boolean**: The boolean type
- **Hashtable**: Hashtable type;
- **List**: The List type:

- **PropertyMap**: Similar to the java *Properties* class. Can be used to load and store java-style properties.
- **Object:** The object type can represent any type.

4.7 Files

File statements are declaration of an output device for text. It uses the keyword 'file'. The file name and extension is given as a parameter. It may also include the relative or absolute path of the output. If no path or a relative path is given, the (externally defined) root directory property is used to define the absolute path of the output.

```
file (c.name + ".java")
file ("c:\tmp\" + c.name + ".java")
file f2 ("test.java")
f2.println ("\t\t output to file f2")
```

Output statements (prints and escaped output) will be written to the latest declared file in the runtime stack. A file declaration is active as long as the declaring rule is active.

When a file is declared in one transformation rule, it will be target for output provided also in rules invoked from the declaring rule, unless the invoked rules declare their own file output. The declared file reference, however, is not visible in invoked rules.

4.8 Print statements

Print statements provide output to an output device, which is either a file or 'standard output'.

```
println ("public class" + c.name);
```

If no file is declared, standard output is used as output. If standard output should be forced, a print should be prefixed with 'stdout'.

```
stdout.println ("public class" + c.name);
```

A couple of other utility print functions are defined, to provide easier whitespace management: `newline` (or `nl`), `tab`, or `space`, followed by an optional count integer. Standard String escape characters (`\n\t`) are also legal within String literals.

```
print ("This is a standard print statement " + aVar.name)
newline (10)
tab(4) ' More escaped output \n\n '
println (" /** Documentation output */ ");
```

4.9 Escaped output

Escaped output provides a different and in some cases simpler way of providing output to a device. Escaped output works similar to most scripting languages, such as Java script.

Escaped output is signaled by escape characters, beginning and ending of an escape. Basically, it is a print statement that can subsume multiple lines and be combined with all expressions that evaluate to a string. Escaped text is signaled by

the characters ‘’’ to start an escape and ‘’’ to end an escape. Note that all whitespace is copied to the output device.

```
‘
   public class ‘ c.name ‘ extends Serializable {
’
```

Note that it is also possible to signal escaped output with ‘<%’ for starting and ‘%>’ signaling the end of the output (this is for historic and backward compatibility reasons).

```
<%
   public class %> c.name <% extends Serializable {
%>
```

4.10 Iterators

Iterators in MOF Script are used primarily for iterating collections of model elements from a source model. The f*orEach* statement defines an iterator over a collection of something, such as a model element collection, a list/hashtable, or a String/Integer.

A forEach statement may be restricted by a type constraint (`collection->forEach (c:someType)`), where the type can be a metamodel type or a built-in type. If a type constraint is not given, all elements in the input collection applies. A forEach statement may also have a guard (an addition constraint), which basically is any kind of Boolean expression. A constraint is described after the type using a vertical bar symbol (‘|’) (`collection->forEach (a:String | a.size() = 2)`

```
-- applies to all objects in the collection of type Operation
c.ownedOperation->forEach(o:uml.Operation) {
      -- statements.
}
-- applies to all objects in the collection
-- of type Operation that has a name that starts with ‘a’
c.ownedOperation->forEach(o:uml.Operation | o.name.startsWith(“a”))
{
    /* statements */
}
// applies to all operation elements in the collection that // has
more than zero parameters and a return type
c.ownedOperation->forEach(o:uml.Operation | o.ownedParameter.size()
> 0 and o.returnResult.size() > 0) {
    /* statements */
}
```

Iterators for List and Hashtable variables:

Iterators may also be defined for List/Hashtable variables, as illustrated below.

```
  var list1:List
  list1.add("E1")
  list1.add("E2")
  list1.add(4)
  list1->forEach(e){
    stdout.println (e)
  }
```

Iterators for Strings

String iterators define loops over the character contents of a string.

```
var myVar: String = "Jon Oldevik"

myVar->forEach(c)
  stdout.print (c + " ")
```

Iterators for Integers

Integer iterators define loops based on the size of the context integer. E.g. integer '3' will produce a loop running 3 times.

```
property aNumber:Integer = 34
aNumber->forEach(n)
     stdout.print(" " + n)
```

Iterators for String and Integer literals

Iterators can also be defined using String or Integer literals. These work the same manner as iterators based on String and Integer properties/variables.

```
"MODELWare, the MDA(tm) project"->forEach(s)
     stdout.print ("" + s)

5->forEach(n)
     stdout.println (" " + n)
```

4.11 Conditional statements

Conditional statements are standard 'if'-statements. They are defined by a single 'if'-branch, followed by a set of 'else-if'-branches, and a possible 'else'-branch.

Arguments to the if/else-if-branches are Boolean expressions.

A conditional statement takes a logical expression as argument.

```
if (c.hasStereoType ("entity")) {
   // statements
} else if (c.hasStereoType("service")) {
   // statements
} else {
   // statements
}

if (c.ownedOperations.size() > 0 and c.name.startsWith("C")) {
       // statements
} else {
       // statements
}
```

4.12 While statements

The while statement works in the same manner as it does in e.g. Java. The keyword is followed by a constraint which can be any kind of Boolean expression, for instance:

```
var  i : Integer = 10
while (i > 0){
   //Statements
   i -=1
}
```

The above while loop will iterate thru the statements nine times before it ends it execution.

4.13 Select expressions

A select expression queries a model collection (or a collection variable) and returns a list containing the result of the query.

Select expressions can (currently) only be used in variable or property assignments. The syntax of the select is similar to that of *forEach.* It takes a type parameter and may have a constraint.

```
var xList:List = self.eClassifiers->select(c:ecore.EClass)
'Number of classes: ' xList.size()
xList->forEach(clazz:ecore.EClass) {
  '\n \t Class: ' clazz.name
}


var yList:List = self.eClassifiers->select(c:ecore.EClass |
c.name.startsWith("MOF"))
```

4.14 Logical Expressions

Logical expressions are expressions that evaluates to true or false and are used in iterator statements and conditional statements.

```
self.ownedAttribute->forEach(p : uml.Property |
    p.association != null){
      // statements
}

if (self.name = ”Car” or self.name = ”Person”) {
}
```

4.14.1 Expression grammar

```
Expression = LogicalExpression | ComparisonExpression |
ValueExpression
LogicalExpression : (LogicalExpression) | not Expression  |
Expression and Expression |
Expression or Expression
ComparisonExpression : ValueExpression {…=<>!=…}
                       ValueExpression
ValueExpression: SimpleExpression | SimpleExpression +
    ValueExpression
SimpleExpression: Literal | Reference | FunctionCall
```

4.15 Transformation inheritance

A transformation may extends another transformation using the *extends* keyword. Only single inheritance is allowed.

The *sub transformation* inherits all rules of the *super transformation*, may override these and call the rules of the *super* using the *'super'* keyword. The example below illustrates.

```
import ("TestInheritanceSuper.m2t")

texttransformation TestInheritanceSub (in ecmodel:ecore) extends
TestInheritanceSuper {
       ecmodel.EPackage::main() {
         self.printMe()
       }

       ecmodel.EPackage::printMe() {
         stdout.println ("TestIneritanceSub::printMe<")
         super.printMe();
         stdout.println ("TestIneritanceSub::printMe>")
       }
}
```

4.16 Abstract rules

MOFScript supports definition of abstract rules. This may be useful in cases of refinement using rule overriding. The example below defines an abstract rule for the metamodel element *Element* from the uml metamodel.

```
 abstract uml.Element::uml2ecore ()
```

4.17 Rule overriding

Transformation rules in MOFScript can override other rules, either from imported transformations or within the same transformation. This has two possible effects:

- Overriding a rule with a new rule with a different context type (the metaclass it applies to), will have the effect that different rules with the same name will be called depending on the metatype. This will have a kind of polymorphic effect with respect to the context type.
- Overriding a rule from an imported transformation with a new one with the same signature. This will merely ensure that calls to that rule will be to the overriding one.
- Overriding a rule from a super transformation in a sub transformation (inheritance overriding). The sub transformation rule will be called instead of the super one. The specializing rule may invoke the rule of the super transformation by using the ‘super’ keyword.

-

```
uml.Package::uml2ecore () {
‘
  <ecore:EPackage name="‘self.name’">
‘
  self.ownedMember->forEach(member:uml.Element)
       member.uml2ecore()
‘
  </ecore:EPackage>
‘
}

uml.Class::uml2ecore () {
  …
}
```

4.18 Invoking External Java Methods

MOFScript has built-in support for invoking external Java code, which enables the integration of external *(black box)* operations from within MOFScript. This is done with the *java* operation.

The syntax is as follows:

```
java (String className, String methodName,
      List/Something parameters, String classpath)
```

The method invoked may be static or class scope. If it is non-static, the class must have a default constructor. The parameters may be *null*, a single parameter (e.g. a String, an integer etc) or a List of parameters if the method takes several parameters.

```
println ("Java: " + java ("org.test.MyTestClass", "myTestString1",
null, "c:/Working/TestJava/"))

var l:List
stdout.println ("Testing Java integration")
l.add("a ")
l.add("b ")
println ("Java: " + java ("org.test.MyTestClass", "myTestString2",
l, "c:/Working/TestJava/"))
```

4.19 Reflection operations on model elements

4.19.1 Reflection ops for EObjects

4.19.1.1 _getFeature

In some cases, a metamodel contains features that conflicts with the keywords in MOFScript. In these cases, a special construct can be used to gain access to that feature, the *'_getFeature("feature name")'* operation.

Using this operation, the conflicting features can be access without compilation errors.

4.19.1.2 _setFeature (name, val)

_setFeature sets the value of a named feature to a given value

4.19.1.3 _getContainer

Gets the container of the context Eobject

4.19.1.4 _getClass

Gets the eClass of the context Eobject

4.19.1.5 _getCrossReferences

Gets the cross references of the context Eobject

4.19.1.6 _getUsages

Gets the usages of the context Eobject

4.19.1.7 _getUsageFeatures

Gets the features that represent the usages of the context Eobject

4.19.1.8 _getContainingFeature

Gets the features contained within the context EObject

4.19.1.9 _delete

Attempts to delete an eobject (corresponds to EcoreUtil.delete(eobj)), which deleted an object from its container and from any where it is referenced.

4.19.2 Reflection ops for EClasses

4.19.2.1 _getOperations

4.19.2.2 __getReferences | _getEReferences

4.19.2.3 _getEReference | _getEFeature

4.19.2.4 _getTypeParameters

4.19.2.5 _getSuperTypes

4.19.2.6 _getFeatures

4.19.3 Reflection ops for EClassifiers

4.19.3.1 _getPackage

Gets the package for a given classifier

4.19.3.2 _isInstance

Checks if an object is an instance of the classifier

4.19.3.3 _eResource()

Gets the associated EResource from an EObject.

4.20 Built-in operations

This chapter summarizes the MOFScript built-in operations.

4.20.1 String operations

- substring (lower : int, upper : int) : String
  - returns the substring from index lower to index upper
- subStringBefore (beforeString: String) : String

    - returns the substring of this string occurring before the 'beforeString'
- subStringAfter (afterString: String) : String
    - returns the substring of this string occurring after the 'afterString'
- toLower () : String
    - converts the string to lower case
- toUpper () : String
    - converts the string to upper case
- firstToUpper () : String
    - converts the first letter of the string to upper case
- firstToLower () : String
    - converts the first letter of the string to lower case
- size () : int
    - returns the size of the string
- indexOf (indexStr : String) : int
    - returns the index of the first occurrence of the 'indexStr' or -1 if it does not exist.
- endsWith (str : String) : Boolean
    - returns true if the string ends with 'str', else false
- startsWith (str : String) : Boolean
    - returns true if the string starts with 'str', else false
- trim () : String
    - removes all trailing and leading white space
- normalizeSpace () : String
    - Trims the string and replaces all sequences of white space characters with a single space.
- replace (replaceWhat : String , withWhat : String) : String
    - replaces all occurrences that matches the regular expression 'replaceWhat' with the 'withWhat' string
- equals (str : String) : Boolean
    - returns true if the string is equals to 'str', else false
- equalsIgnoreCase (str : String) : Boolean
    - returns true if the string is equal to the string 'str' ignoring casing
- isUpperCase (int index) : Boolean
    - returns true if character at position 'index' is upper case. If no index is given, first position (index=0) is used.
- isLowerCase (int index) : Boolean
- returns true if character at position 'index' is lower case. If no index is given, first position (index=0) is used.
- charAt (int index) : String
- returns the character (as a String) at position 'index'.
- forEach ()
- Iterator operation for Strings. Iterates over each character in the string.
- matches (regexp) : Boolean
- Checks if the string matches the regular expression 'regexp'.
- first (int i) : String
    - gets the first i chars of the string
- last (int i) : String

- gets the last I chars of the string
- strtok (String regexp) : String
- returns a tokenized version of the String, like C++ strtok
- strcmp (String other): Integer
- Compares two Strings
- isAlpha () : Boolean
- Checks if the string consists of letters
- isAlphaNum () : Boolean
- checks if the string consists of letters and numbers

Example:

```
"myString".toLower()
c.name.size()
c.name.endsWith("Fa")
```

4.20.2 Integer operations

The integer operations

- Standard arithmetic operations: +, -, *, /
- forEach ()
    - Iterator operation for integers. Iterates over the size of the integer (from 0 to its size).

4.20.3 List Operations

The list operations

- add (e:Object) : Boolean
- adds an object to the list
- remove (e:Object) : Boolean
- removes an object from the list.
- size () : Integer
- returns the size of the list
- clear () : void
- empties the list
- first () : Object
- returns the first element of the list
- last () : Object
- returns the last element of the list
- isEmpty () : Boolean
- returns true if the list is empty (size == 0), false otherwise
- forEach () [iterator operation]
- iterator mechanism applied on lists.
- select (x : type | condition)
- selects a subset of objects from a collection
- addAll (list)
- appends the objects in the 'list' to the end of the current list
- addAllFirst (list)
- appends the objects in the 'list' to the start of the current list

- addBefore (item, list)
- appends the objects in the 'list' before a given item already in the list.
- addAfter (item, list)
- appends the objects in the 'list' after a given item already in the list.
- indexOf (element) : Integer
- returns the index of a given element

4.20.4 Hashtable Operations

The Hashtable operations

- put (key: Object, value: Object)
- puts an element 'value' with the 'key' into the hashtable
- get (key: Object)
- returns the value associated with the 'key' parameter
- size () : Integer
- returns the size of the hashtable (i.e. number of 'key' elements)
- clear () : void
- empties the hashtable
- keys () : List
- returns the list of keys in this hashtable
- values () : List
- returns the list of values in this hashtable
- first () : Object
- returns the first object in the hashtable
- last () : Object
- returns the last object in the hashtable
- isEmpty () : Boolean
- returns true if the list is empty (size == 0), false otherwise
- forEach () [iterator operation]
- Iterator mechanism applied in the hashtable – the iterator will iterator the values of the hashtable.

4.20.5 PropertyMap Operations

A *PropertyMap* is a specialisation of Hashtable.
In addition, it offers the following operations to load and store properties:

- load ("prop-file")
- loadXML("xml-prop-file")
- store ("prop-file")
- storeXML("xml-prop-file")

4.20.6 Model Collection Operations

- size () : int
- returns the size of the collection
- first () : Object
- returns the first object of the collection
- isEmpty (): Boolean

- checks if the collection is empty
- forEach () [Iterator operation]
- Iterator mechanism applied on the model collection
- select (x : type | condition)
- selects a subset of objects from a collection

Example:

```
if (c.attributes.size() = 0) {
   stdout.println ("Size is 0")
}
c.attributes->forEach (p:uml.Property | p = c.attributes.first()) {
    stdout.println ("First attribute")
}
```

4.20.7 Model Operations

- objectsOfType (type) : Collection
- Returns all instances of type 'type' within a given model object
- store (file uri)
- Stores the given model to the given file uri.

Example:

```
ec.objectsOfType (ec.EClass)->forEach (eccl) {
   …
}
```

4.20.8 OCL Operations

- oclIsTypeOf (type: typeRef): Boolean
- returns 'true' if the type in question is exactly the same as the input type, false otherwise.
- oclIsKindOf (type: typeRef) : Boolean
- returns 'true' if the type in question is the same or a subtype of the input type, false otherwise
- oclGetType () : String
- returns the name of the type in question

4.20.9 System / utility operations

System and utility operations

- position () : Integer
- Returns the index counter value of context forEach loop, the position of the current elements in the loop. Returns -1 if there is no loop.
- count () : Integer
- Returns the index counter value of the nearest context forEach loop, taking filters into account. Returns -1 if there is no loop.
- getenv (String property): String
- Gets an environment variable. Equivalent to Java *System.getProperty().*

- setenv (String property, String value)
- Sets an environment variable. Equivalent to Java *System.setProperty().*
- time () : String
- Returns the current time as a String. The only format currently supported is HH:MM:SS
- date () : String
- Returns the current date as a String. The only format currently supported is DD/MM/YY

4.20.10 UML2 Operations

The following operations are available when UML2 models are loaded and the UML2 Eclipse plug-in is available.

- Boolean hasStereoType (String stereotypeName)
- List<Stereotype> getAppliedStereotypes ()
- Stereotype getAppliedStereotype ()
- Boolean hasValue (Stereotype stObj | String stName, String valueName)
- Object getValue (Stereotype stObj | String stName, String valueName)

They are all applicable to UML 2 modelelements, such as classes. E.g:

```
     self.ownedElements -> forEach (c:uml.Class) {
        if (c.hasStereoType (“myStereoType”) {
            ‘Class stereotype: ’ + c.name
            if (c.hasValue (“myStereoType”, “myProperty”)) {
               ‘\t Stereotype property: ‘ + c.getValue
(“myStereoType”, “myProperty”)
      }
     }
```

4.21 Unprotected Blocks

MOFScript supports the notion of unprotected blocks. These blocks are created with the use of the unprotect keyword in the transformation code, as illustrated in the transformation code for operations below.

```
self.ownedOperation->forEach(o:uml.Operation){
    '\n  'o.visibility' void ' o.name'(){'
    unprotect{
       '    //User code here for operation'
    }
    '  }\n'
}
```

The resulting code, shown below, represents the unprotected block as comments containing a *#BlockStart* and a *#BlockEnd* and an identifier for the source model element.

```
  public void printAuthor(){
//#BlockStart number=4 id=_MeMJULEPEdu-Vepu7rgPLg
  //User code here for operation
```

```
//#BlockEnd number=4
  }
```

Between the block comments, the user can insert or remove code, and the changes will be preserved the next time the transformation is run. All the traces that have references to the file after the block will also be generated in accordance with their new position in the file.

4.22 MOFScript Aspect extension

A prototype implementation of aspects for MOFScript is in place. This is preliminary implementation and by no means complete. For example, there is not a lot of semantics checking of an aspect.

The MOFScript aspects work by inserting transformation code into (a copy of) the target transformation, which is stored as a new transformation. I.e. the weaving is all done compile time, not run time.

4.22.1 The aspect

An aspect is a specialization of a transformation. It contains pointcuts and advices, and may also define ordinary MOFScript rules.

An aspect is defined as a separate aspect transformation, identified by the keyword ‘aspect’

```
aspect JavaAspect {
 …
}
```

4.22.2 Pointcuts

Pointcuts identify points of execution in the MOFScript transformation (joinpoints).

A pointcut has a name and may have a type specification. Currently, two types of pointcuts are defined in MOFScript:

- call: Refers to the calling of a specific set of rules
- execute: Refers to the rules themselves.

A pointcut further defines a match critera in terms of a regular expression (as a string literal). The match critera defines what rules are matched by the pointcut.

```
pointcut classPrivateAttributeCall(Class) call
("classPrivateAttribute");
pointcut propertyNamedCall call ("property.*");
pointcut propertySetterExecute execute ("propertySetter")
```

Currently, there is no type checking of the aspects. The type passed as parameter for a pointcut must be a model type name without any prefix.

4.22.3 Advice

Advice describes the actions to be executed at when specific joinpoints (specified by the pointcuts) occur. An advice refers to one pointcut and has three possible modifiers: *before, after, and around*, which gives the semantics of what happens to the advice actions.

- Before: Inserts the advice code before the code identified by the pointcut.
- After: Inserts the advice code after the code identified by the pointcut.
- Around: Replaces the code identified by the pointcut.

```
aspect JavaAspect {
  pointcut propertyGetterCall call ("propertyGetter");
  pointcut propertySetterExecute execute ("propertySetter")
  pointcut propertyNamedCall call ("property.*");

  before propertyNamedCall {
    log ("\n calling a property function")
  }

  before propertyGetterCall {
    'code inserted before call to property getter'
  }

  around propertySetterExecute {
    '
    public void set ' self.name 'Replaced('self.type.name') {
    }
    '
  }
}
```

4.22.4 Executing the aspect

An aspect is executed (in the Eclipse IDE) by running just as an ordinary MOFScript transformation. As input for the aspect is another MOFScript transformation.

The implementation of the aspect transformation is itself as a MOFScript transformation.

4.23 Model to Model Transformation

MOFScript can be used for model-to-model transformations. This can be used to create models from scratch, or augment existing models. The following describes the features supporting model-to-model capabilities.

4.23.1 Declaring output parameters

A transformation can declare output parameters, implying that these represent output models of the transformation.

```
texttransformation M2T2Mofscript (in m2t:"http://omg.org.MOF2Text",
      out mofscript: http://org.eclipse.mofscript.model
(mofscript.MOFScriptSpecification)) {
```

```
}
```

An output declaration has a name, a model URI, and a specification of the model element type which is the root if the output model (in the example this is *mofscript.MOFScriptSpecification).*

4.23.2 Using output models from transformations (in Java)

When a transformation is executed from Java, its output models can be accessed with the *getOutputModels()* method on the ExecutionManager.

```
execMgr.executeTransformation();
Collection output = execMgr.getOutputModels();
Object resultModel = null;
if (output.size() > 0) {
   resultModel = output.toArray()[0];
}
```

4.23.3 Creating instances – the *new* operator

New model elements can be created using the 'new' operator.

```
var newVar:mofscript.VariableDeclaration = new
                        mofscript.VariableDeclaration();
```

The new operator creates a new instance of the specified type. A comma-separated list of parameters on the form <name=value> can be sent in the create call, which results in these properties being set on the created object.

```
… …new mofscript.VariableDeclaration(name="MyVariable");
```

4.23.4 Setting property values

Simple property values are set by standard assignment:

```
newVar.name = "YourVariable"
newVar.value = "Your Value"
```

4.23.5 Adding values to collection properties

Two main operations are used to add objects to feature collections: add and addOrg.

add adds a copy of the original object to the collection

addOrg adds by reference, i.e. the reference to the original object is added.

```
mst.variables.addOrg (newVar)
```

Unless you want to add object copies, you want to use *addOrg* rather than *add.*

4.23.6 Cloning an object

Objects can be cloned by calling the operation **clone()**. Cloning can be used on any object structure and results in a deep copy of the object structure. This can be used for copying an input model, modifying it, and later storing it.

4.23.7 Storing an object

Models are stored using the **store("file name")** operation on a model object. This operation can be executed on either output or input models.

4.23.8 Obtaining and storing an EResource

It is possible to obtain the EResource for an EObject using the operation o._eResource().

var res : ecore.EResource = self._eResource

The resource can be modified by using its *getContents() method.*

res.getContents().add(obj);

The resource can be stored by invoking *store()* on the resource, either with a name, or without any parameters to replace the original resource.

res.store("newresource.ecore");

res.store ();

5 The MOFScript debugger

The MOFScript Debugger is a light-weight add-on to the existing MOFScript runtime. It does not extend/use the Eclipse debugging framework.

Debugging is available through the pop-up menu. It allows to set / remove break point and to start/stop/step debugging.

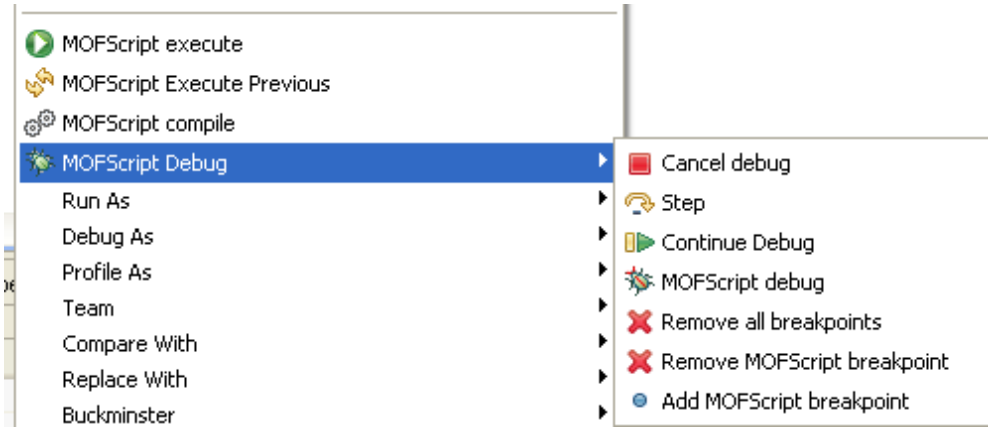


Figure 12 Context Menu for Debugging

When debugging, MOFScript execution will halt at specified break points. You can step through execution using control keys:

- F6 steps one statement
- F8 continues until next break point is reached.

A debug view (MOFScript Debug View) provides a simple view of variable state.

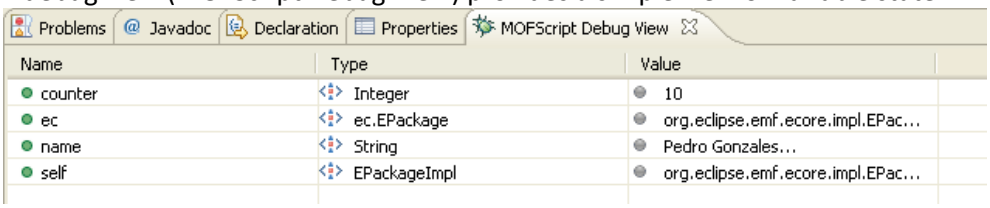


Figure 13 MOFScript Debug View

Choosing 'Cancel Debug' will disable the debug session and continue normal execution.

6 Integrating with MOFScript Java API

The code below illustrates integration with MOFScript using the Java API.

```
import java.io.File;
import java.util.Iterator;
import org.eclipse.emf.common.util.URI;
import org.eclipse.emf.ecore.EObject;
import org.eclipse.emf.ecore.resource.Resource;
import org.eclipse.emf.ecore.resource.ResourceSet;
import org.eclipse.emf.ecore.resource.impl.ResourceSetImpl;
import org.eclipse.emf.ecore.xmi.impl.XMIResourceFactoryImpl;
import org.eclipse.mofscript.MOFScriptModel.MOFScriptSpecification;
import org.eclipse.mofscript.parser.MofScriptParseError;
import org.eclipse.mofscript.parser.ParserUtil;
import org.eclipse.mofscript.runtime.ExecutionManager;
import org.eclipse.mofscript.runtime.ExecutionMessageListener;
```

```
import org.eclipse.mofscript.runtime.MofScriptExecutionException;
import org.eclipse.uml2.uml.UMLPackage;


public class TestAPI implements ExecutionMessageListener {

	private ParserUtil parserUtil = null;
	private ExecutionManager execMgr = null;

	/**
	 * Constructor
	 */
	public TestAPI () {
		UMLPackage lePackage = UMLPackage.eINSTANCE;
		ParserUtil parserUtil = new ParserUtil();
		ExecutionManager execMgr =
ExecutionManager.getExecutionManager();
	}


	/**
	 * Parses a transformation
	 * @return numebr of parse errors
	 */
	protected int parse (String transformation) {
		File f = new File (transformation);
		MOFScriptSpecification spec = parserUtil.parse(f,
true);
		int errorCount =
ParserUtil.getModelChecker().getErrorCount();
		// check for errors:
		Iterator errorIt =
ParserUtil.getModelChecker().getErrors(); // Iterator of
MofScriptParseError objects
		System.out.println ("Parsing result: " + errorCount + "
errors");
		if (errorCount > 0) {

			for (;errorIt.hasNext();) {
				MofScriptParseError parseError =
(MofScriptParseError) errorIt.next();
				System.out.println("\t \t: Error: " +
parseError.toString());
			}
		}
		return errorCount;
	}

	/**
	 * Executes the transformation
	 * @param inputModel - the name (path) of the inputmodel
	 */
	protected void execute (String inputModel) {
		XMIResourceFactoryImpl _xmiFac = new
XMIResourceFactoryImpl();
		EObject sourceModel = null;
		File sourceModelFile = new File (inputModel);
		ResourceSet rSet = new ResourceSetImpl ();

	rSet.getResourceFactoryRegistry().getExtensionToFactoryMap().
put("*", _xmiFac);
		URI uri =
URI.createFileURI(sourceModelFile.getAbsolutePath());
```

```
		Resource resource = rSet.getResource(uri, true);

		if (resource != null) {
			if (resource.getContents().size() > 0) {
				sourceModel = (EObject)
resource.getContents().get(0);
			}
		}

		// set the source model for the exeution manager
		execMgr.addSourceModel(sourceModel);
		// sets the root output directory, if any is desired
(e.g. "c:/temp")
		execMgr.setRootDirectory("");
		// if true, files are not generated to the file systsm,
but populated into a filemodel
		// which can be fetched afterwards. Value false will
result in standard file generation
		execMgr.setUseFileModel(false);
		// Turns on/off system logging
		execMgr.setUseLog(false);
		// Adds an output listener for the transformation
execution.

	execMgr.getExecutionStack().addOutputMessageListener(this);

		try {

			execMgr.executeTransformation();
			// execMgr.getOutputModels();
		} catch (MofScriptExecutionException mex) {
			mex.printStackTrace();
		}

	}

	/**
	 * Parsing and executing
	 */
	public void test (String transformation, String inputModel) {

		int errs = parse(transformation);
		if (errs == 0) {
			execute(inputModel);
		}
	}

	/**
	 * ExecutionMessageListener interface operations
	 */
	public void executionMessage (String type, String
description) {
		System.out.println (type + " - " + description);
	}

	/**
	 * Main
	 * @param args
	 */
	public static void main (String[] args){
		final String transformation = "UMLTest.m2t";
		final String inputModel = "ServiceModel.uml";
		TestAPI api = new TestAPI ();
```

```
            api.test(transformation, inputModel);
        }
}
```